

Лекция №2. Языки описания аппаратуры. Часть 1

Толкачев Максим

m.tolkachev@metrotek.ru

- I. Что такое аппаратура;
- II. Какие языки описания аппаратуры есть;
- III. Причины существования несинтезируемых подмножеств HDL языков;
- IV. SystemVerilog: структура модуля;
- V. SystemVerilog: комбинационная и последовательная логика;
- VI. Запускаем симулятор.

I. Что такое аппаратура

ПЛИС не исполняет программы!

- Программа

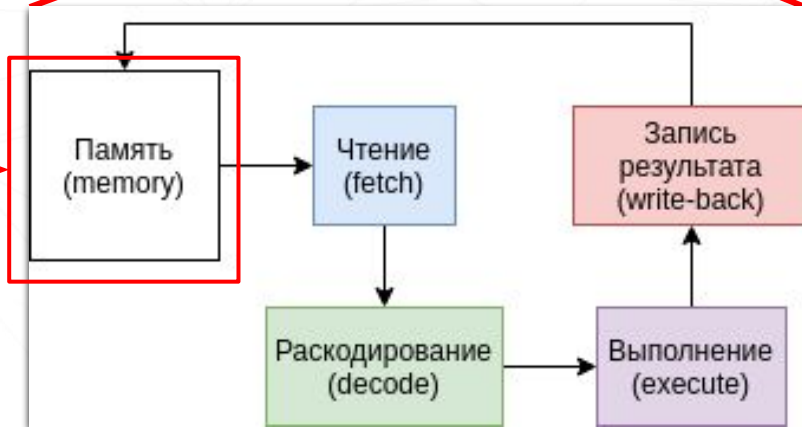
```
int main() {
    int x=10,y=15;
    return 0;
}
```



```
0000000000000000 <main>:
0: 55          push    %rbp
1: 48 89 e5    mov     %rsp,%rbp
4: c7 45 fc 0f 00 00 00 movl    $0xf,0x4(%rbp)
b: b8 00 00 00 00 mov     $0x0,%eax
12: 5d          pop     %rbp
17: c3          retq
```



- Упрощенный конвейер процессора



II. Какие языки описания аппаратуры есть

HDL -- Hardware Description Language	High-level HDL	HLS -- High-level Synthesis
AHDL	BlueSpec (Haskell/SystemVerilog)	Intel High-Level Synthesis Compiler (C++)
VHDL	Clash (Haskell)	Xilinx Vivado High-Level Synthesis (C/C++/SystemC)
Verilog	MyHDL (Python)	Mentor Catapult HLS (C/C++/SystemC)
SystemVerilog	Chisel 3 (Scala)	High-Level Synthesis with LegUp

HDL -- Hardware Description Language	High-level HDL	HLS -- High-level Synthesis
--------------------------------------	----------------	-----------------------------

- | | | |
|---|---|---|
| <ul style="list-style-type: none"> ● Не требует генерации; ● cycle accurate. | <ul style="list-style-type: none"> ● Генерирует Verilog; ● cycle accurate. | <ul style="list-style-type: none"> ● Генерирует Verilog; ● Уровень выше RTL (не cycle accurate). |
| <ul style="list-style-type: none"> ● Не только для FPGA, но и для описания других схем и процессов (ASIC); ● Чтобы схема работала, нужно писать “правильно”. И этой информации нет в стандарте языка, зависит от вендора. | <ul style="list-style-type: none"> ● Как HDL, но дополнительные абстракции; ● Дополнительные проверки и ограничения помогающие разработчику; ● Очень быстрая функциональная симуляция. | <ul style="list-style-type: none"> ● Time to market; ● Переносимость кода между платформами; ● Абстракция над интерфейсами (легко менять интерфейсы); ● Очень быстрая функциональная симуляция. |

- VHDL:
 - + Сильная типизация;
 - + Пользовательские типы;
 - + Широко распространен (особенно в старых проектах).
 - Строгая типизация вызывает много неудобств и скорее вредит;
 - Приходится писать много лишнего кода;
 - Мало функций для симуляции.
- Verilog:
 - + Понятнее программистам (сравнительно с VHDL);
 - + Компактный код (сравнительно с VHDL);
 - + Широко распространен.
 - Слабая типизация, отсутствие пользовательских типов;
 - Нет пользовательских типов;
 - Нет удобных средств для верификации;

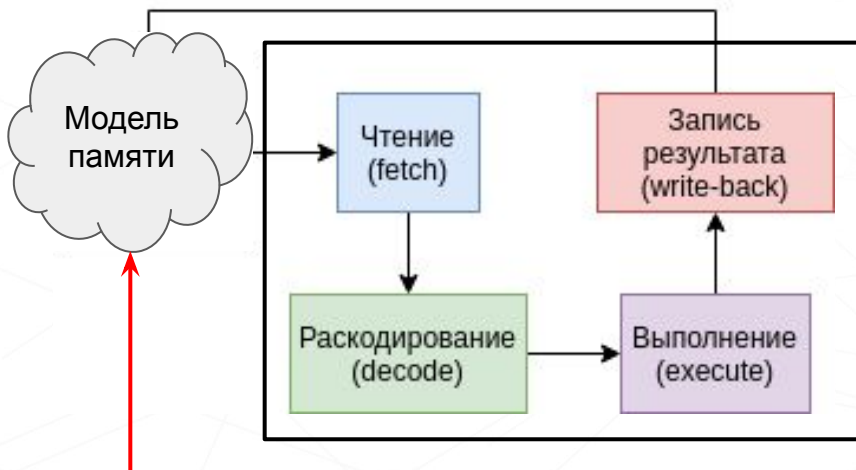
- SystemVerilog (расширение Verilog):
 - + Пользовательские типы
 - + Огромное несинтезируемое подмножество для верификации
 - + Удобно писать легко читать
 - + Поддержка **во многих** средствах проектирования:
- Слабая типизация (легко что-то не туда присвоить);
- Сложно получить “скелет” и начать отлаживать;
- Много условностей, как можно, а как нельзя использовать конструкции.

Производитель ПЛИС	Среда проектирования	Поддержка SV
Xilinx	ISE (старые семейства)	
	Vivado	
Intel	Quartus	
MicroSemi	Libero (Synplify Pro)	
Lattice	Diamond (Synplify Pro)	

III. Причины существования несинтезируемых подмножеств HDL языков

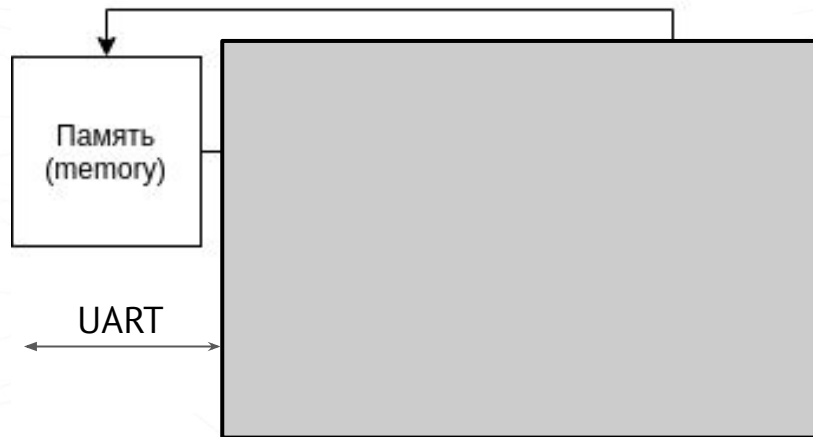
Схему нужно отлаживать

Отладка в симуляции:



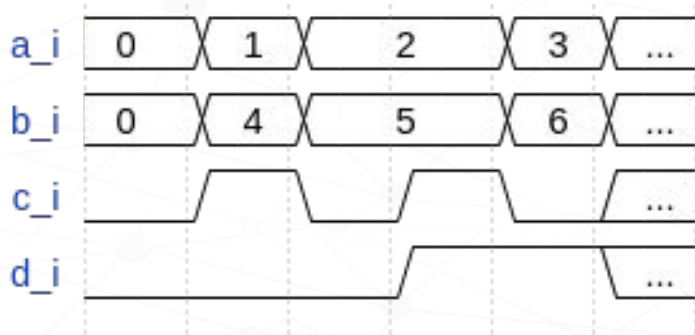
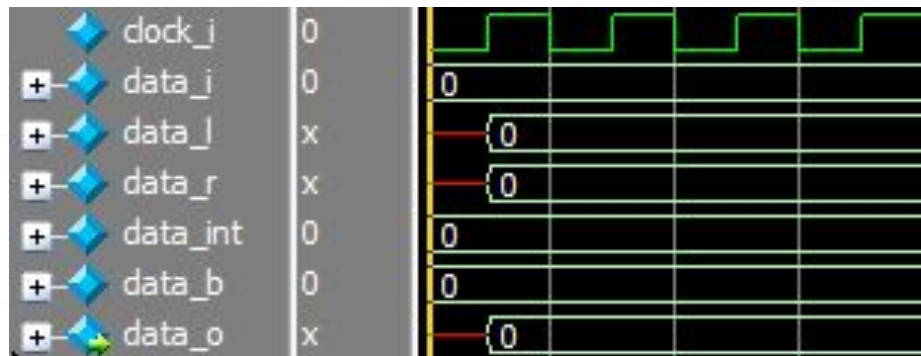
- Разные варианты задержки записи и чтения;
- Специально сгенерированные последовательности команд;
- Отладка на временных диаграммах состояния каждого из блоков на каждую инструкцию.

Работа в железе:

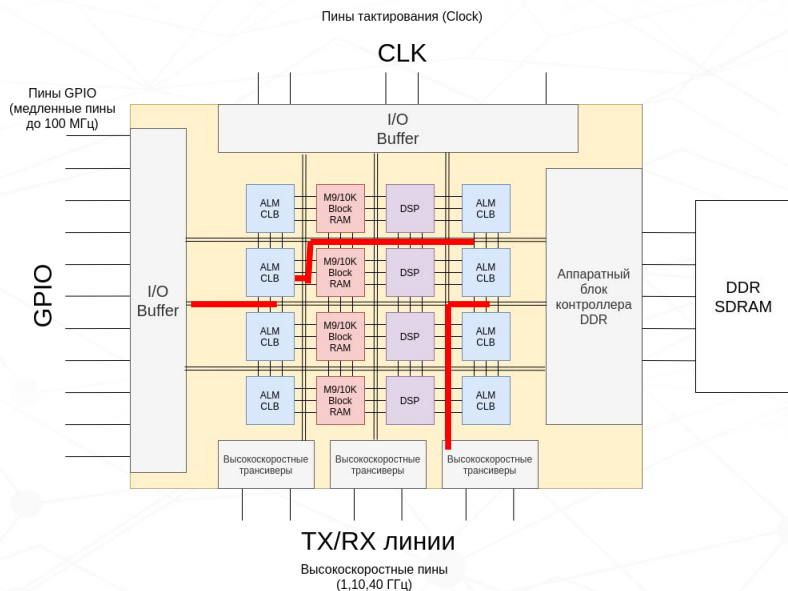
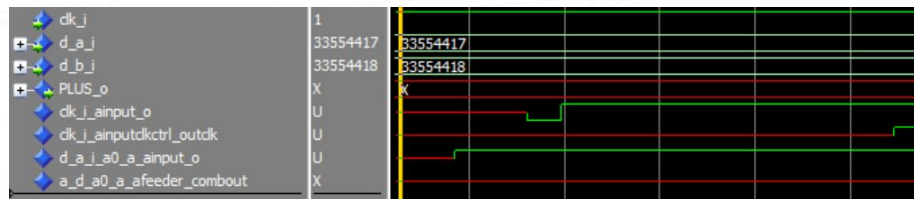


- Возможности отладки ограничены;
- Тяжело специально воспроизводить последовательность воздействий;
- Много неизвестных в системе.

Функциональная симуляция



Временная симуляция



IV. SystemVerilog: структура модуля

SystemVerilog. Структура кода

Название модуля

Текстовый файл:
address_counter.sv

```
module address_counter (
  input      clock_i,
  input      reset_i,
  output [31:0] address_o
);

  logic [31:0] address_cnt;

  always_ff @(posedge clock_i)
    if( reset_i )
      address_cnt <= 0;
    else
      address_cnt <= address_cnt + 1;

  assign address_o = address_cnt;
endmodule
```

Входные и
выходные
сигналы

Внутренние
сигналы

Процедурный
блок
(procedural block)

Присваивание
(continuous
assignment)

Разрядность сигнала:

- 31 -- номер старшего бита
- 0 -- номер младшего бита

Список чувствительности

```

module delay_1_tick (
    input          clock_i,
    output logic [31:0] data_i,
    output logic [31:0] data_o
);
    // 4 state:
    logic [31:0] data_l;
    reg [31:0] data_r;

    // 2 state:
    int data_int;
    bit [31:0] data_b;

    always_ff @( posedge clock_i )
    begin
        data_l  <= data_i;
        data_r  <= data_i;
        data_int <= data_i;
        data_b  <= data_i;
    end
    assign data_o = data_l;
endmodule
    
```

2-state: 0 1	4-state: 0 1 X Z
bit	logic
int	reg
byte	wire
longint	integer
...	...

Состояние **X** -- не синтезируемое. В железе примет случайное значение или 0 или 1.

Состояние **Z** -- синтезируется только для управления выходными пинами (переводит их в высокоимпедансное состояние).

```
module delay_1_tick (
    input      clock_i,
    output logic [31:0] data_i,
    output logic [31:0] data_o
);
    // 4 state:
    logic [31:0] data_l;
    reg [31:0] data_r;

    // 2 state:
    int data_int;
    bit [31:0] data_b;

    always_ff @( posedge clock_i )
    begin
        data_l  <= data_i;
        data_r  <= data_i;
        data_int <= data_i;
        data_b  <= data_i;
    end
    assign data_o = data_l;
endmodule
```




```

module test_module;
  int a,b;

  initial
  begin
    a = 1;
    b = a;
  end

  always_comb
  begin
    a = 2;
    b = a;
  end

  always_ff @( posedge clock )
  begin
    a <= 3;
    b <= a;
  end

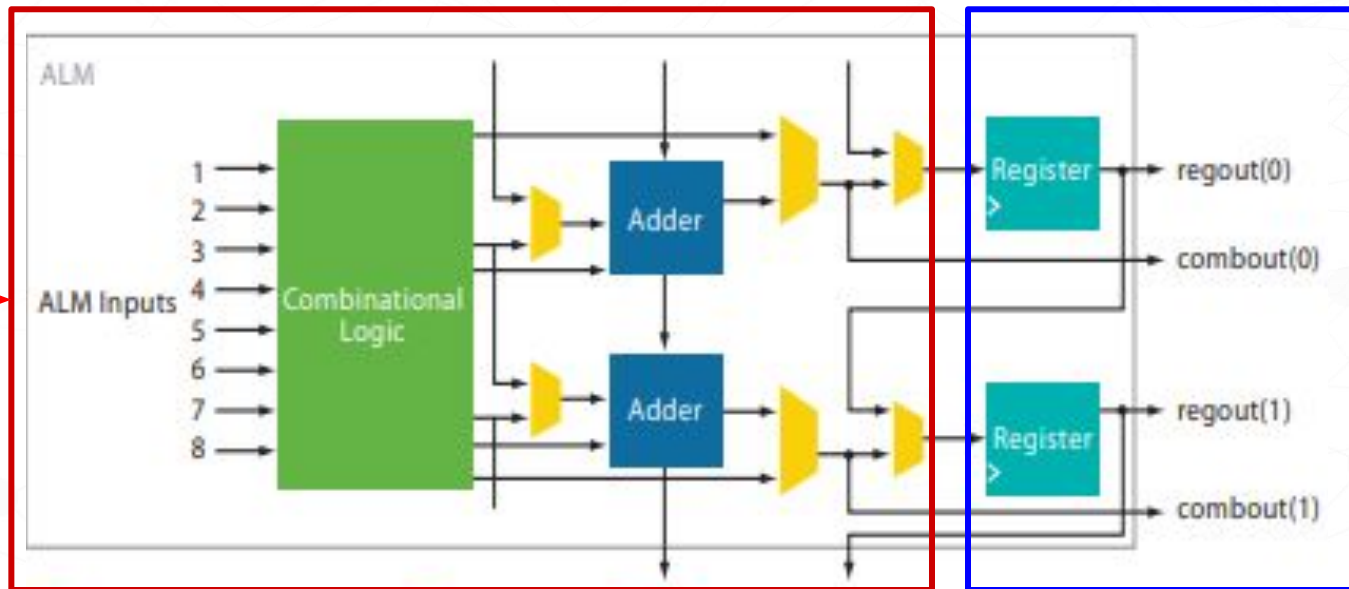
  assign a = 4;

endmodule

```

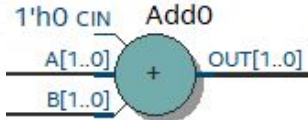
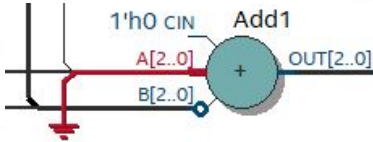
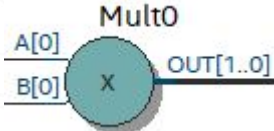
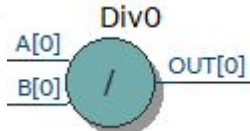
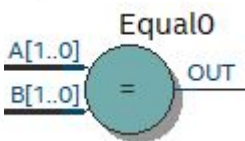
<code>initial</code>	Запускается при старте симуляции, работает 1 раз.
<code>always_comb</code>	Запускается при любом изменении сигналов, используемый в блоке.
<code>always_ff</code>	Запускается по событию в списке чувствительности.
<code>assign</code>	“Соединение” сигналов.


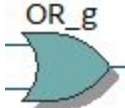
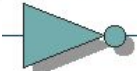
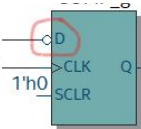

V. SystemVerilog: комбинационная и последовательная логика



Комбинационная логика: +, -, AND, OR, NOT
Работает очень быстро (можно считать мгновенно).

Последовательная логика: сохранение в регистр
Данные защелкиваются на фронте тактового сигнала.

Операция	Обозначение на схеме RTL	Оператор SystemVerilog
Сложение		+
Вычитание		-
Умножение		*
Деление		/
Сравнение на равенство		==

Операция	Обозначение на схеме RTL	Оператор SystemVerilog
AND	<p>AND_g</p> 	&&
OR	<p>OR_g</p> 	
NOT	<p>NOT_o</p>  	!
XOR	<p>XOR_g~0</p> 	^

```

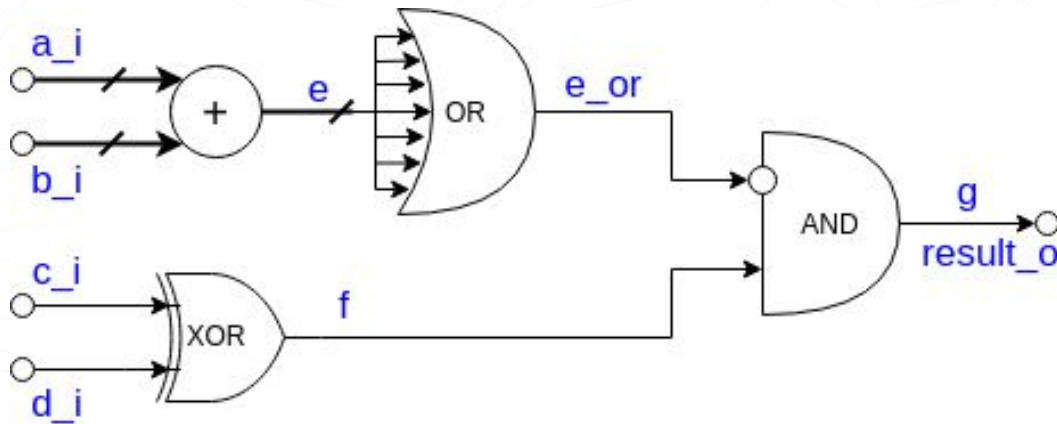
module logic (
    input [7:0]  a_i,
    input [7:0]  b_i,
    input        c_i,
    input        d_i,
    output logic result_o
);

    logic [8:0] e;    // a + b
    logic        e_or; // bitwise or of e
    logic        f;    // c xor d
    logic        g;    // f and not e_or;

    always_comb
    begin
        e      = a_i + b_i;
        e_or   = |e;
        f      = c_i ^ d_i;
        g      = f && !e_or;
        result_o = g;
    end

endmodule

```



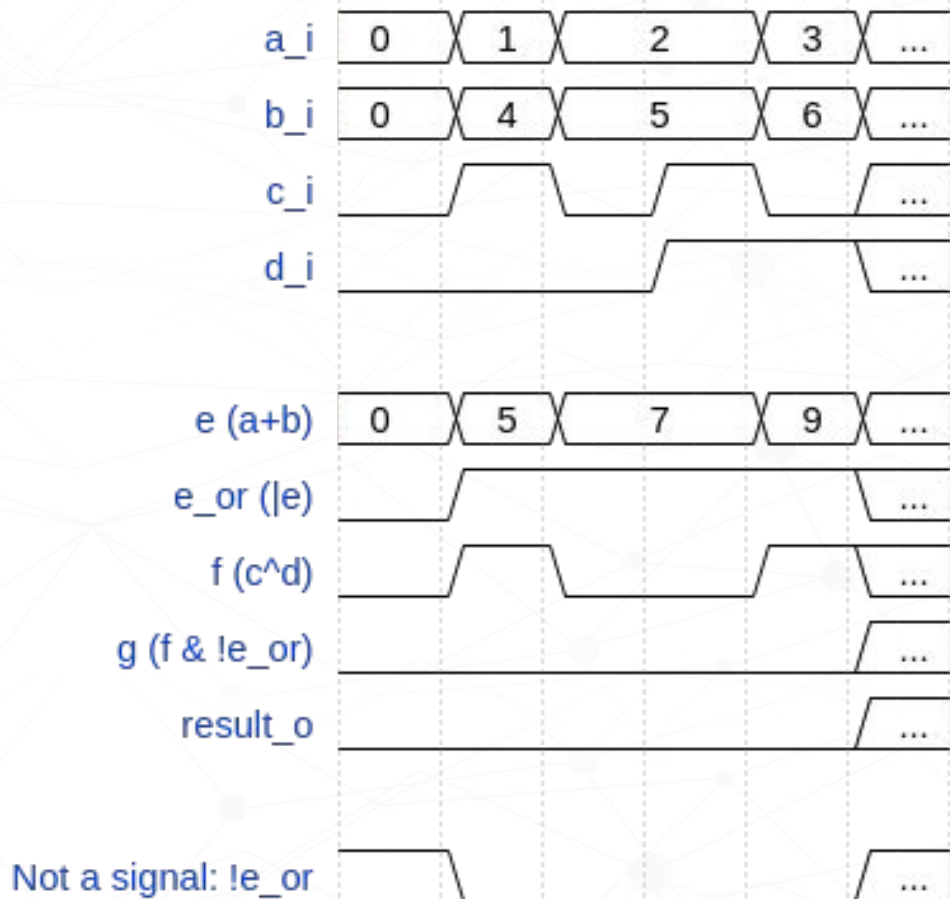
```

module logic (
    input [7:0] a_i,
    input [7:0] b_i,
    input      c_i,
    input      d_i,
    output logic result_o
);

    logic [8:0] e;    // a + b
    logic      e_or;  // bitwise or of e
    logic      f;     // c xor d
    logic      g;     // f and not e_or;

    always_comb
    begin
        e      = a_i + b_i;
        e_or   = |e;
        f      = c_i ^ d_i;
        g      = f && !e_or;
        result_o = g;
    end

endmodule
    
```



```

module logic (
    input [7:0] a_i,
    input [7:0] b_i,
    input      c_i,
    input      d_i,
    output logic result_o
);

logic [8:0] e;    // a + b
logic      e_or; // bitwise or of e
logic      f;    // c xor d
logic      g;    // f and not e_or;

always_comb
begin
    e      = a_i + b_i;
    e_or   = |e;
    f      = c_i ^ d_i;
    g      = f && !e_or;
    result_o = g;
end

endmodule
    
```

==

```

module logic (
    input [7:0] a_i,
    input [7:0] b_i,
    input      c_i,
    input      d_i,
    output logic result_o
);

logic [8:0] e;    // a + b
logic      e_or; // bitwise or of e
logic      f;    // c xor d
logic      g;    // f and not e_or;

always_comb
begin
    g      = f && !e_or;
    result_o = g;
end

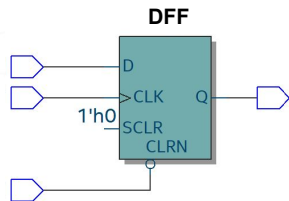
assign e = a_i + b_i;

assign e_or = |e;

always_comb
    f = c_i ^ d_i;
endmodule
    
```


Что внутри ПЛИС. Регистр

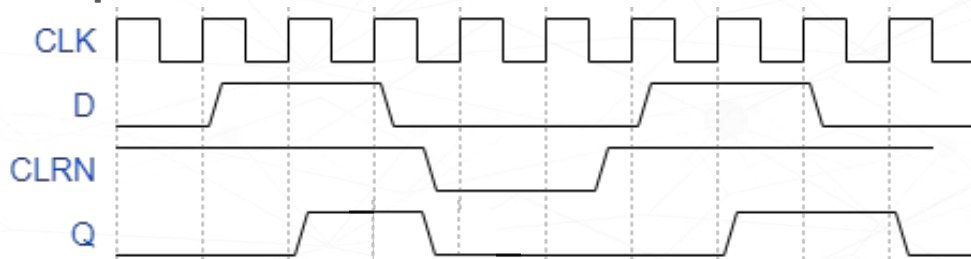
- Регистр: триггер, D-триггер, D flip-flop, DFF:



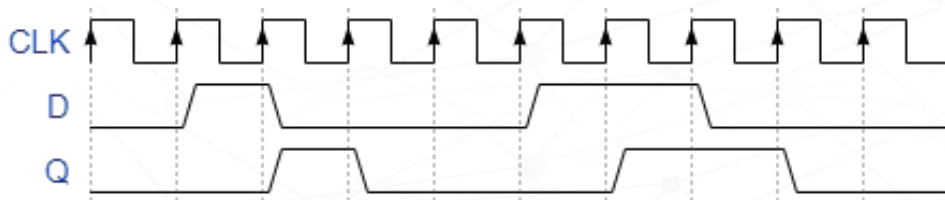
Inputs			Output
CLR N	CLK	D	Q
0	X	X	0
1	⌋	0	0
1	⌋	1	1
1	0	X	Q _o
1	1	X	Q _o

Q_o (Q old): Старое значение **Q**

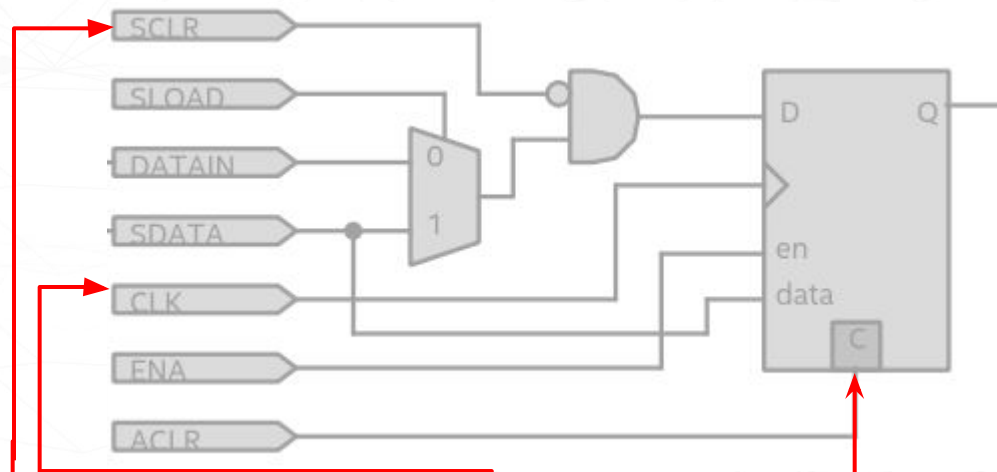
Сброс:



CLR N = 1. Рабочее состояние:

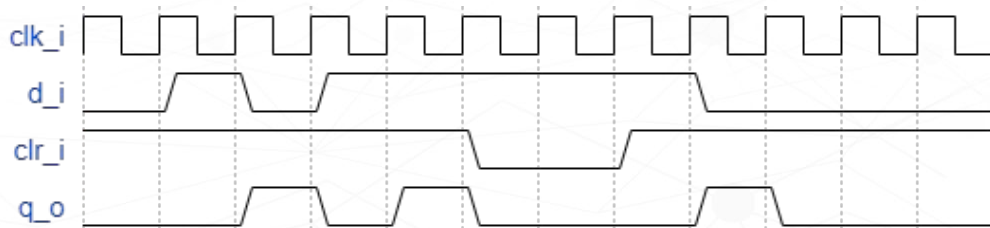


```
always_ff @(posedge clk_i or negedge clr_i)
begin
    if( !clr_i )
        q_o <= 1'b0;
    else
        if( sync_reset )
            q_o <= 1'b0;
        else
            q_o <= d_i;
end
```



```
always_ff @(:posedge clk_i or negedge clr_i)
begin
    if( !clr_i )
        q_o <= 1'b0;
    else
        if( sync_reset )
            q_o <= 1'b0;
        else
            q_o <= d_i;
end
```

```
always_ff @(posedge clk_i or negedge clr_i)
begin
    if( !clr_i )
        q_o <= 1'b0;
    else
        if( sync_reset )
            q_o <= 1'b0;
        else
            q_o <= d_i;
end
```

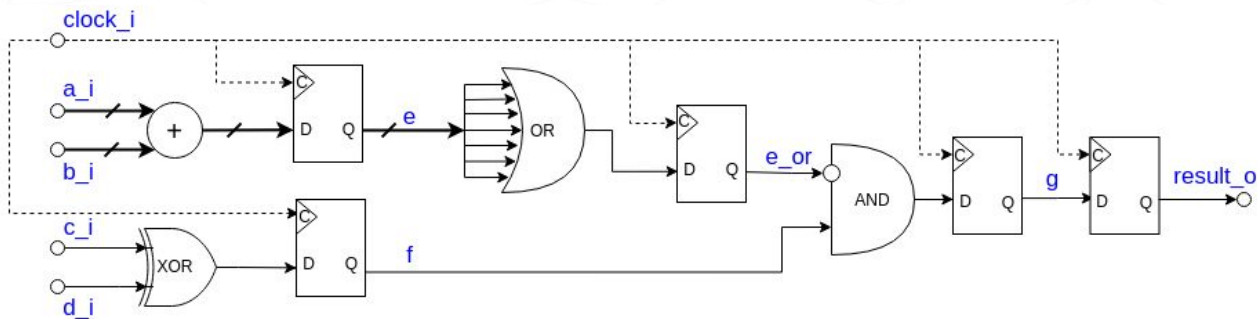


```
module logic_with_d (
    input      clock_i,
    input [7:0] a_i,
    input [7:0] b_i,
    input      c_i,
    input      d_i,
    output logic result_o
);
```

```
    logic [8:0] e;
    logic      e_or;
    logic      f;
    logic      g;
```

```
    always_ff @(posedge clock_i)
    begin
        e      <= a_i + b_i;
        e_or   <= |e;
        f      <= c_i ^ d_i;
        g      <= f && !e_or;
        result_o <= g;
    end
```

```
endmodule
```



```
module logic (  
    input [7:0] a_i,  
    input [7:0] b_i,  
    input      c_i,  
    input      d_i,  
    output logic result_o  
);  
  
logic [8:0] e;    // a + b  
logic      e_or; // bitwise or of e  
logic      f;    // c xor d  
logic      g;    // f and not e_or;  
  
always_comb  
begin  
    e      = a_i + b_i;  
    e_or   = |e;  
    f      = c_i ^ d_i;  
    g      = f && !e_or;  
    result_o = g;  
end  
  
endmodule
```

```
module logic_with_d (  
    input      clock_i,  
    input [7:0] a_i,  
    input [7:0] b_i,  
    input      c_i,  
    input      d_i,  
    output logic result_o  
);  
  
logic [8:0] e;  
logic      e_or;  
logic      f;  
logic      g;  
  
always_ff @(posedge clock_i)  
begin  
    e      <= a_i + b_i;  
    e_or   <= |e;  
    f      <= c_i ^ d_i;  
    g      <= f && !e_or;  
    result_o <= g;  
end  
  
endmodule
```

Блокирующие и неблокирующее присваивание

Блокирующие:

```
logic [2:0] D;
```

```
initia  
begin
```

2 1
D = a_i + b_i;

4 3
E = D + c_i;

```
end
```

a_i	1
b_i	2
c_i	3
D	3
E	6

Неблокирующие:

```
logic [2:0] D;
```

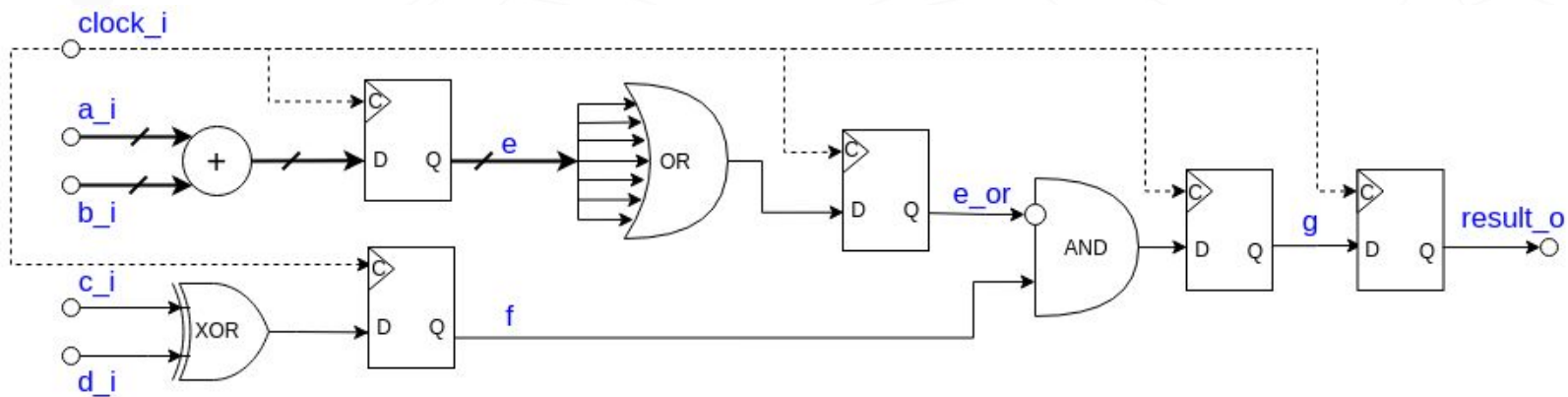
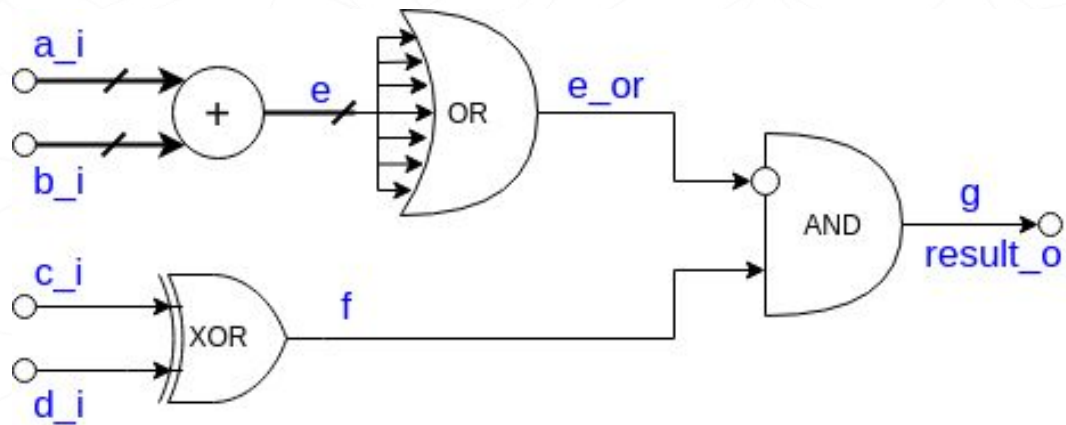
```
initia;  
begin
```

3 1
D <= a_i + b_i;

4 2
E <= D + c_i;

```
end
```

a_i	1
b_i	2
c_i	3
D	3
E	X



```

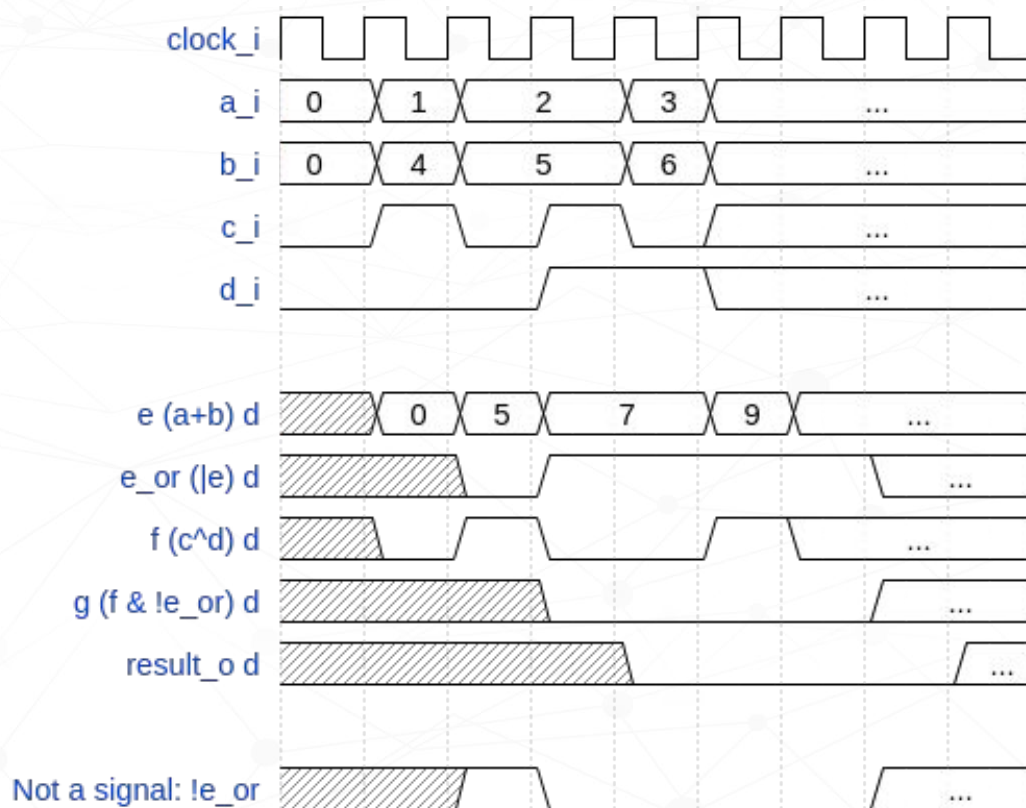
module logic_with_d (
    input      clock_i,
    input [7:0] a_i,
    input [7:0] b_i,
    input      c_i,
    input      d_i,
    output logic result_o
);

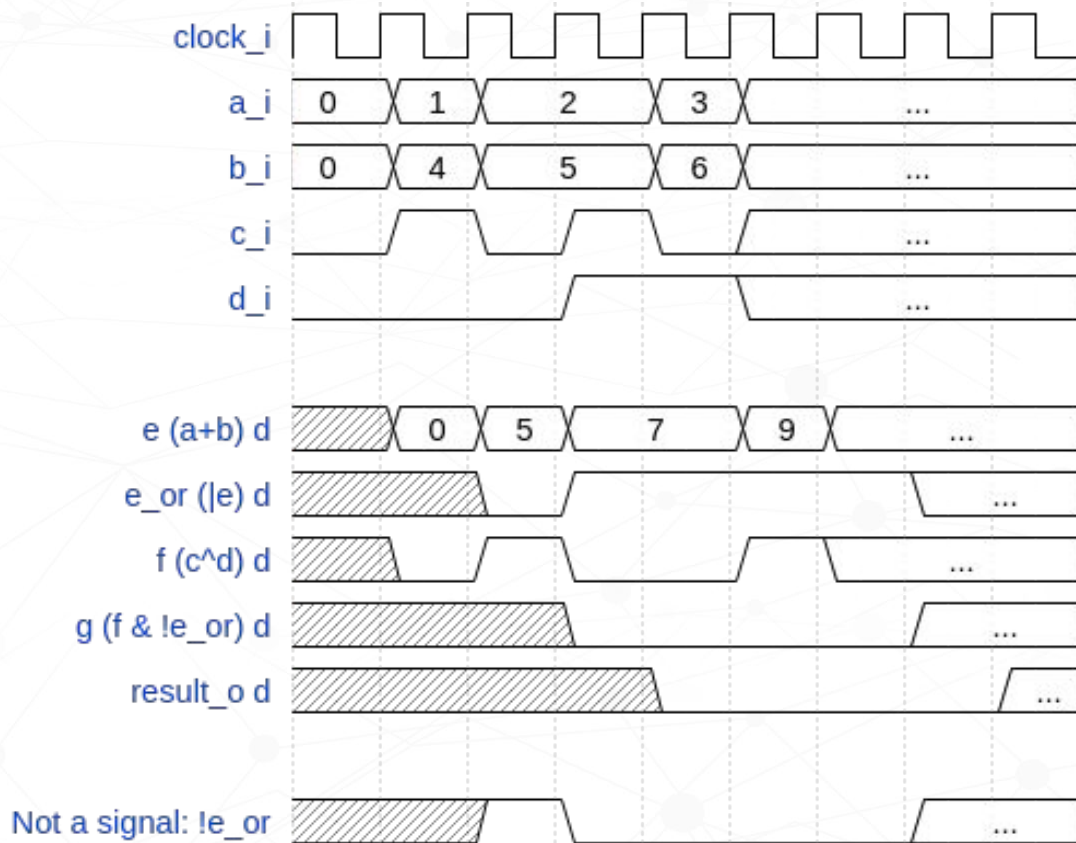
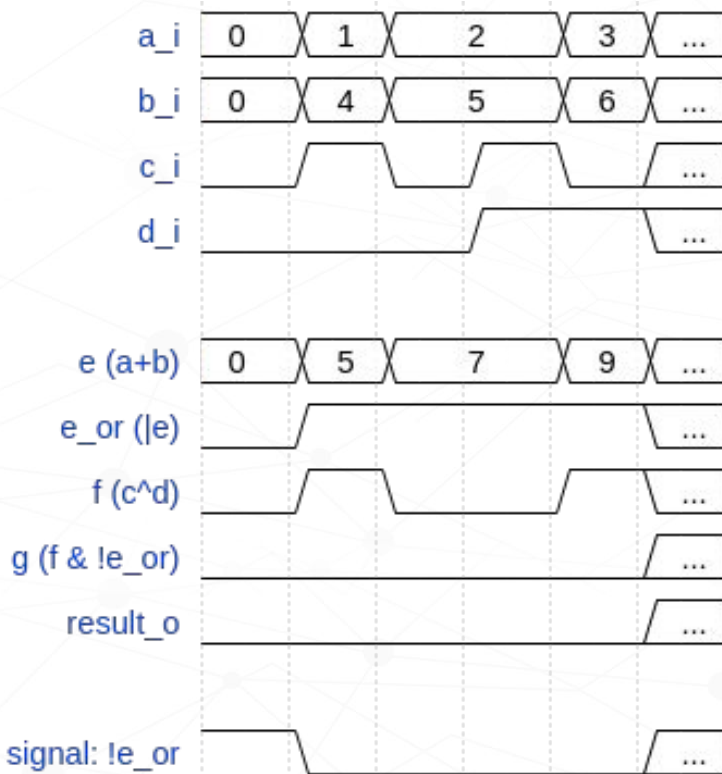
    logic [8:0] e;    // a + b
    logic      e_or;  // bitwise or of e
    logic      f;     // c xor d
    logic      g;     // f and not e_or;

    always_ff @(posedge clock_i)
    begin
        e      <= a_i + b_i;
        e_or   <= |e;
        f      <= c_i ^ d_i;
        g      <= f && !e_or;
        result_o <= g;
    end

endmodule

```





```

module logic_with_d (
    input      clock_i,
    input [7:0] a_i,
    input [7:0] b_i,
    input      c_i,
    input      d_i,
    output logic result_o
);

logic [8:0] e;    // a + b
logic      e_or; // bitwise or of e
logic      f;    // c xor d
logic      g;    // f and not e_or;

always_ff @(posedge clock_i)
begin
    e      <= a_i + b_i;
    e_or   <= |e;
    f      <= c_i ^ d_i;
    g      <= f && !e_or;
    result_o <= g;
end

endmodule

```

==

```

module logic_with_d (
    input      clock_i,
    input [7:0] a_i,
    input [7:0] b_i,
    input      c_i,
    input      d_i,
    output logic result_o
);
logic [8:0] e;    // a + b
logic      e_or; // bitwise or of e
logic      f;    // c xor d
logic      g;    // f and not e_or;

always_ff @(posedge clock_i)
    e      <= a_i + b_i;

always_ff @(posedge clock_i)
    e_or   <= |e;

always_ff @(posedge clock_i)
begin
    f      <= c_i ^ d_i;
    g      <= f && !e_or;
    result_o <= g;
end

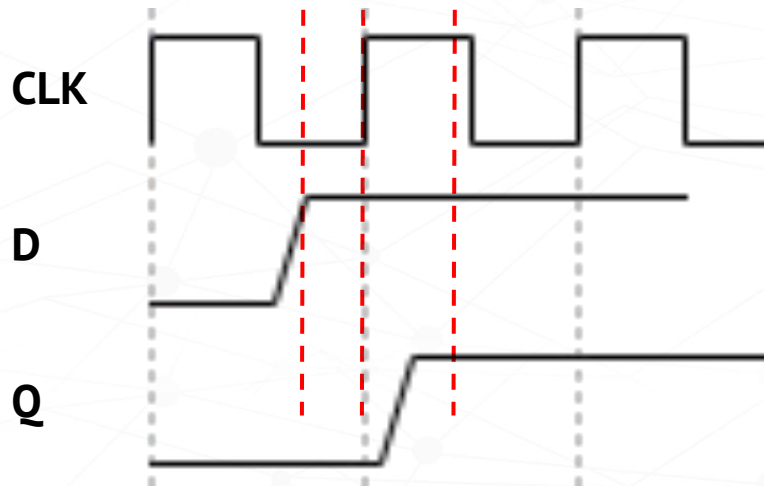
endmodule

```

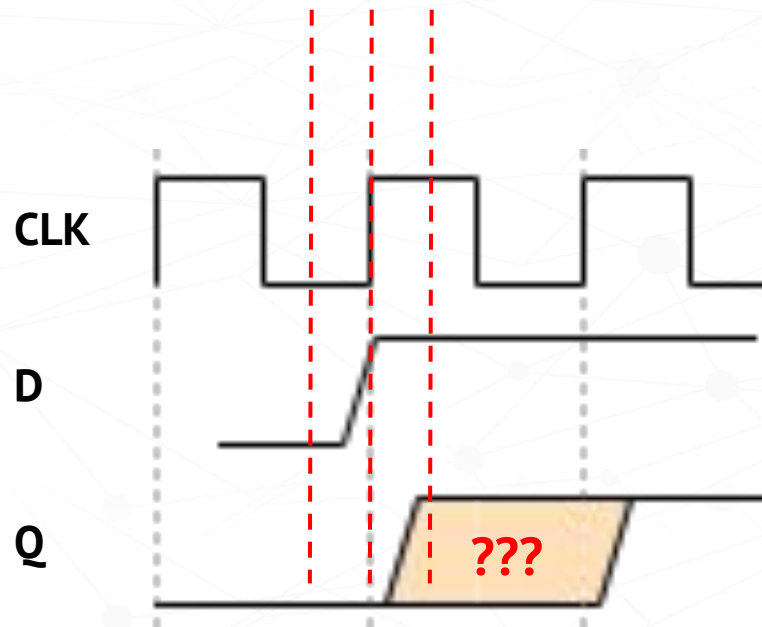
SystemVerilog. Последовательная логика.

Требования к сигналу.

T_{setup} T_{hold}



Все очень плохо!



SystemVerilog. Паттерны для синтеза

Правильно	Неправильно	Что синтезируется
<pre>always_comb a = b + c;</pre> <p>или</p> <pre>assign a = b + c;</pre>	<pre>always_comb a <= b + c;</pre> <p>или</p> <pre>always @(b or c) a = b + c;</pre>	<p>Во всех правильных случаях комбинационная логика (заполнение для LUT).</p> <p>В неправильном варианте могут быть отличия в симуляции и в поведении в железе.</p>
<pre>always_ff @(posedge clk_i) a <= b + c;</pre> <p>или</p> <pre>assign a_tmp = b + c;</pre> <pre>always_ff @(posedge clk_i) a <= a_tmp;</pre>	<pre>always_ff @(posedge clk_i) a = b + c;</pre> <p>или</p> <pre>always_ff @(posedge clk_i) begin a_tmp <= b + c; a <= a_tmp; end</pre>	<p>Чтобы получился регистр и симуляция совпадала с синтезом -- в always_ff блоке только неблокирующее (<=) присваивание!</p> <p>В списке чувствительности, кроме клокa, может быть только асинхронный сброс.</p>

VI. Запускаем симулятор

Файл: **counter.sv**

```
module counter (
    input      clock_i,
    input      reset_i,
    output [3:0] data_o
);

    logic [3:0] cnt;

    always_ff @( posedge clock_i )
        begin
            if( reset_i )
                cnt <= '0;
            else
                cnt <= cnt + 1;
            end

    assign data_o = cnt;
endmodule
```

Файл: **top_tb.sv**

```
module top_tb;
    bit clk;
    bit reset;
    initial
        forever
            #5 clk = !clk;

    logic [3:0] cnt_from_dut;

    counter DUT (
        .clock_i ( clk           ),
        .reset_i ( reset         ),
        .data_o   ( cnt_from_dut )
    );

    initial
        begin
            reset <= 1'b0;
            #99;
            @( posedge clk );
            reset <= 1'b1;
            @( posedge clk );
            reset <= 1'b0;
            $display( cnt_from_dut );
            @( posedge clk );
            $display( cnt_from_dut );
        end
endmodule
```

Файл: **make_sim.do**

```
vlib work  
  
vlog -sv counter.sv  
vlog -sv top_tb.sv  
  
vsim -novopt top_tb  
add log -r /*  
add wave -r *  
run -all
```